Component-Based, Run-Time Flight Software Modification

Mohammad Shahabuddin
818-354-4993
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
shahab@jpl.nasa.gov

Alexander Murray
818-354-0111
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
alex.murray@jpl.nasa.gov

Vanessa Carson
818-353-5503
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
vcarson@caltech.edu

Abstract—Missions involving robotic space flight typically have a way to change the software that controls the flight system, or some part of it, such as an instrument, after launch. Usually this is accomplished by uplinking small sets of binary machine instructions and writing them to known locations in memory. We present an approach, used on the Aquarius mission, that involves replacing running components of, or adding components to, the running software at a higher logical level, specifically at the software architecture level, and on the C++ rather than machine-language level. This approach provides significant advantages in flexibility, robustness, reliability, and testability. We present the component-based flight software (FSW) design features that enable these capabilities. We then discuss the approach used to verify the robustness and reliability of these techniques, and finally describe usages to date. [1] [2]

TABLE OF CONTENTS

1. INTRODUCTION

Robotic space missions need to modify FSW after launch for various reasons: to fix a software bug that is discovered in operations, to accommodate hardware changes or failures that necessitate different FSW behavior, to take advantage of unanticipated science opportunities, and frequently because the FSW was simply not complete at launch.

Modifications can be made to the running software, or to a copy in spacecraft persistent storage, in which case they take effect only when the modified version is loaded during a reset or reboot. In this paper, we focus on modifications to the running software.

Modifications to the running software can be made in most spacecraft flight systems and also in the FSW of instruments that have CPUs. These modifications are usually accomplished by uplinking files containing binary machine instructions, typically a modified version of a routine that is already present in the running FSW, and then overwriting the original version of the function.

Such binary code patches are non-persistent; if the spacecraft or instrument reboots, the original version of the FSW is reloaded into RAM from some form of persistent memory, and the modifications are lost. This is also true of the approach on Aquarius which we present here.

Our FSW architecture is an object-oriented, component-based architecture, implemented in C++. Each component in the design plays a specific role, such as processing all commands and telemetry to/from a particular subsystem, or providing any needed interfaces to a particular piece of hardware. Some components play an infrastructure type of role, such as dispatching commands to other components, or collecting telemetry from other components and formatting it for transmission.

A run-time modification in our approach means either replacing one of the running components with a new version, or adding an entirely new component, one that doesn't fulfill any of the existing roles of the architecture. A replacement component may be only slightly different from the old version, or it can be entirely different in terms of the behavior it exhibits, possibly containing many new classes and functions, and handling new commands, producing new telemetry, or spawning new threads. This flexibility makes FSW modification a tremendously powerful technique for handling unexpected situations.

Our patching approach does not require the ops. team to have knowledge of the layout of the code in the flight computer memory, because we use the operating system's loader to place the new machine code in memory and to resolve references that the new code makes to functions in the existing on-board code base. So the patching process, and the new component itself, can be tested on target machines with different memory layouts or sizes, or even on Unix workstations. This makes the patching process much

[1] 1-4244-1488-1/08/$25.00 ©2008 IEEE.
[2] IEEEAC paper #1450, Version 13, Updated November 27, 2007

more robust and reliable. At least one mission has suffered serious consequences because of poking an incorrect address, and this would be virtually impossible with our approach.

Aquarius is a compound instrument consisting of a radiometer, scatterometer, and command and data handling subsystem, the heart of which is a PowerPC processor that runs the FSW discussed in this paper. The requirements on the software include communicating with the spacecraft through 1553 and high-speed serial buses, commanding, monitoring, and receiving data from the two sub-instruments, radar data reduction, and temperature monitor and control. We present the FSW design in general terms, leaving out any detail that isn't directly related to the modification capability. A detailed discussion of the FSW architecture in general is given in [1]. See [2] for an overview and mission-level details of Aquarius.

The next two sections are aimed at FSW engineers who are interested in understanding the details of how our techniques work. Those more interested in the operational and system aspects of this patching approach may prefer to skip to the Verification or Applications section.

Unified Modeling Language (UML) diagrams are used throughout. The diagrams are fairly intuitive, but readers wanting to understand the fine points of this paper should be familiar with the UML. A concise and readable introduction to the language is given in [4].

2. Modifiable FSW Architecture

Understanding our running FSW modification approach depends not on the details of the Aquarius FSW architecture, but on the concepts and features of the architecture that are driven by the modification requirements. We present a view of the architecture that concentrates on those aspects. We begin with a run-time view of the FSW in operation, shown in Figure 1. In this diagram, we have an Input component that plays a role of routing incoming data (not shown) to a Command component, which then distributes commands to other components. Some of the components produce telemetry, which is sent to the Telemetry component, which then sends some form of the collected telemetry data to the Output component, from which it leaves the system (not shown).

The System component is shown providing CPU time to some of the components, enabling periodic activities in those components. This System component plays a unique and crucial role in the architecture, as we shall see. It's noteworthy that, though this diagram presents an accurate, if abstracted, view of the running Aquarius FSW, it could as well represent any member of a whole family of systems, encompassing many flight systems, or indeed many embedded systems.

Now we move to a similar view of the running system, but introduce some type information, in Figure 2. In this figure, the connections between the components are shown as typed interfaces. The Input component requires an implementation of the interface *MessageSink*, through which the data flows to the Command component, which provides that interface. Commands are distributed to the components that handle them through the *Queue* interface, suggesting that commands may be processed on separate threads, as they are in most cases. Components that produce telemetry implement the *TelemetryProducer* interface, which is required by the Telemetry component to receive the data. The Telemetry component requires an implementation of the *MessageSink* interface, which is provided by the Output component.

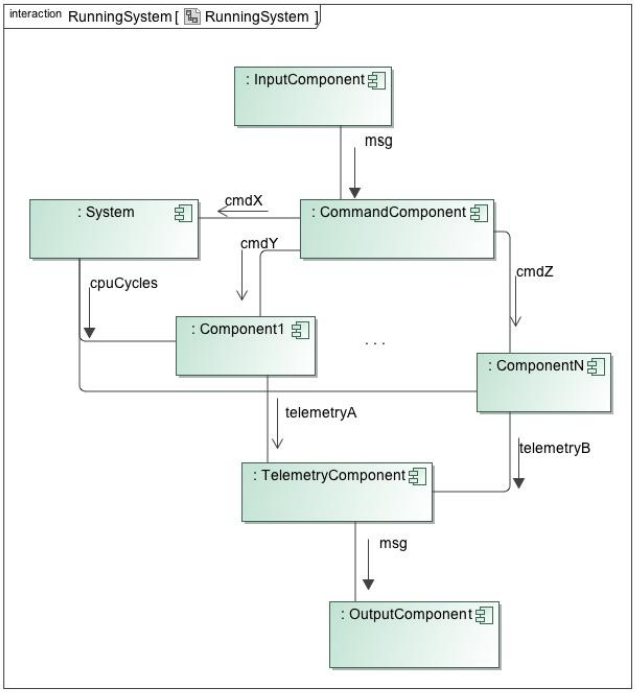

Figure 1 - The Running System

The System component is shown requiring an implementation of the *Clock* interface, which in Aquarius is provided by software that handles an external interrupt at 100 Hz. The Clock interface drives a scheduler in the System component, which in turn distributes CPU time to those components that are connected to the System component through a provided *Periodic* interface implementation.

In order to replace one of these running components, it is necessary first to disconnect the component instance from every other component instance in an orderly way, then to instantiate the new component, and then establish the connections between the new component and other component instances (though it may be that the new version has different types of connections to different components from those of the original version.)

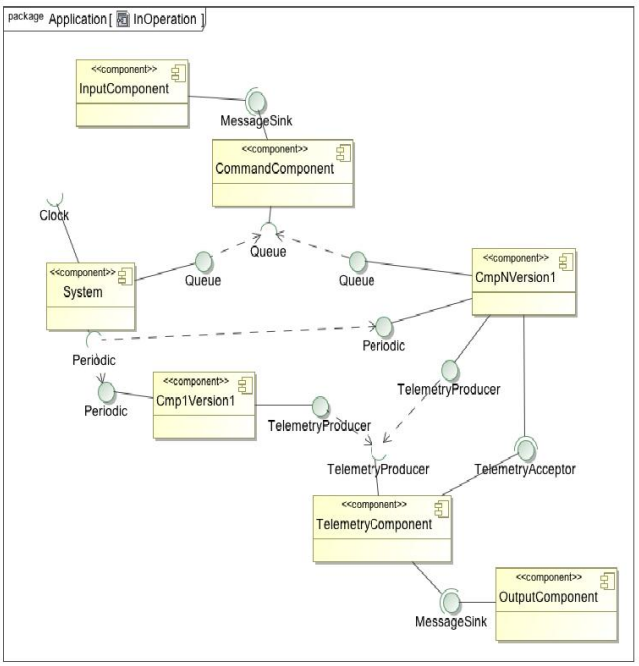

Figure 2 - In Operation Mode

The disconnection and reconnection is accomplished using a special set of interfaces designed for that purpose. These are shown in Figure 3, which shows a different run-time depiction of the same set of components. In this figure, the component with *CmpNVersion1* is shown providing the *TaskOwner*, *CommandProcessor*, *TelemetrySource*, and *Linkable* interfaces.

By providing these interfaces, the component itself provides the logic necessary to connect itself to the rest of the architecture, or to disconnect itself from the architecture. In order to provide these interfaces, the component requires implementations of the *CommandDispatcher*, *Scheduler*, and *TelemetryManager* interfaces.

Our architecture uses two categories of interfaces: operational mode interfaces, such as *TelemetryProducer*, and architectural change mode interfaces, such as *TelemetryManager*. As we discuss each of these groups of interfaces, we refer to these categories of interfaces as *operational* and *architectural*. In our UML diagrams, we show the operational interfaces in green, and the architecture change mode interfaces in aqua.

An important feature of the design is the abstract base class *Component*, from which all other component classes are derived. This *Component* class provides trivial, no-op implementations of all of the architectural interfaces – that is, any calls to the functions implementing the interfaces have no effects, and return a value indicating success. In this way, derived classes are not forced to provide implementations of the architectural interfaces, and must only override the base class' no-op implementation if they need their version of the interface to actually have effects on the architecture. We describe these effects in the following paragraphs.

Command Processing Interfaces

Figure 4 shows the architectural and operational interfaces involved in processing commands. A typical component class, such as *CmpNVersion1* in the diagram, embodies the fact that it can handle commands by implementing the *CommandProcessor* interface, and overriding the trivial, no-op implementation of that interface which is provided by the *Component* base class. If a component doesn't need to handle commands, then it doesn't implement this interface.

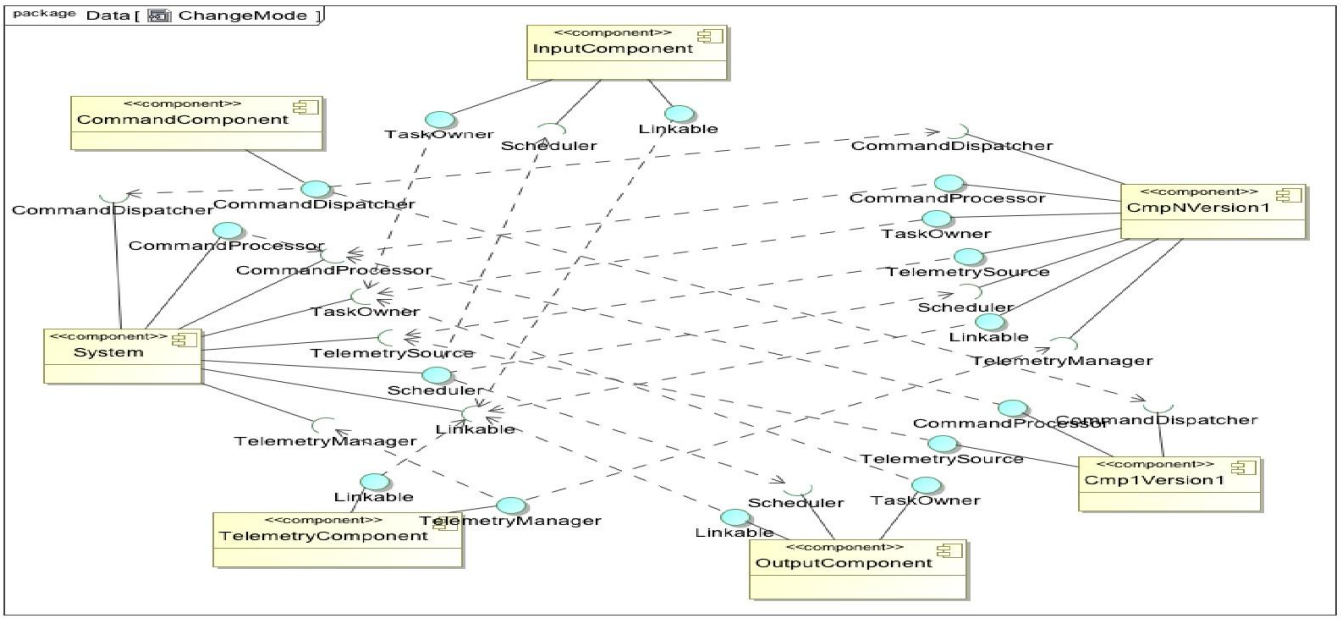

Figure 3 - Architecture Change Mode

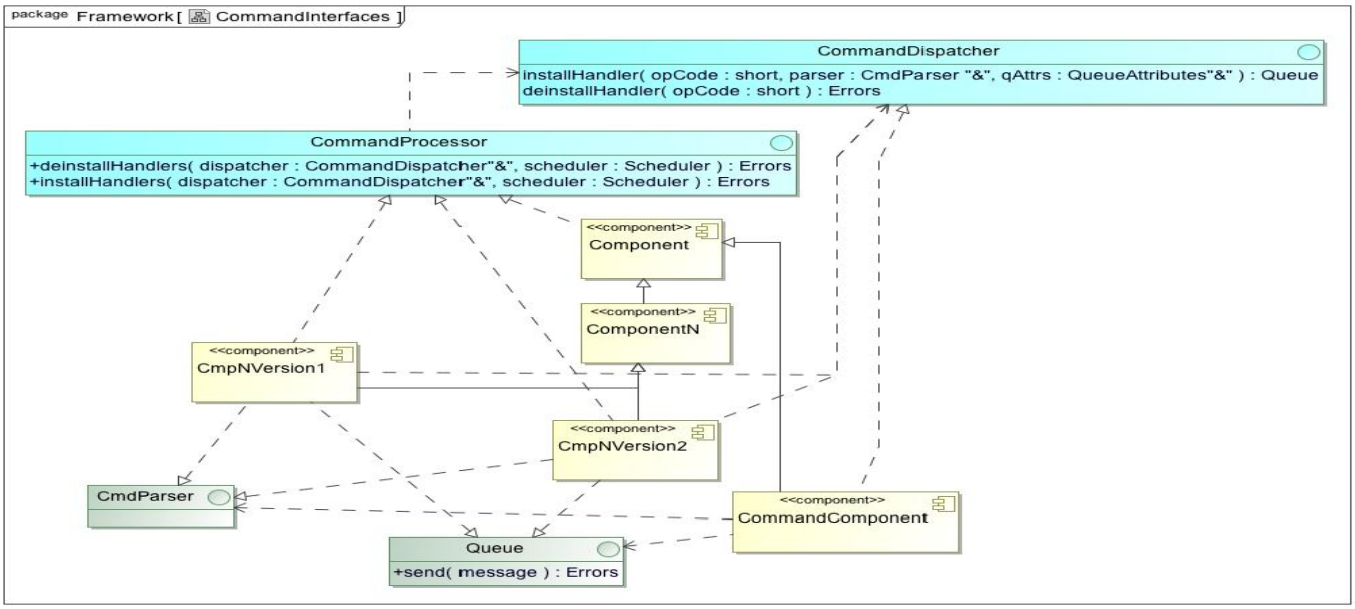

Figure 4 - Command Processing Interfaces and Classes

Note also that each of the components that implement *CommandProcessor* require an implementation of *CommandDispatcher*. When the operation *installHandlers* is called on the component, an implementation of *CommandDispatcher* is provided with the call as one of the arguments. This is an example of an interface implementation hand-off, which is a common pattern among the architectural interfaces. To establish command-processing connections, the System component calls *installHandlers* on the component. Then, for each command that the component wants to process, it calls installHandler on the given CommandProcessor implementation, providing a command op-code, *CommandParser*, and queue parameters. The CommandDispatcher creates an implementation of *Queue* and returns it (if all goes well).

Thus we see that the knowledge of which commands are handled by which components resides in the component classes themselves. This distributed model of information about the architectural connections is a key feature that enables the flexibility of this approach, and helps ensure that components are independent of each other to the extent possible. To be sure, the architectural framework imposes constraints: for example, only one component will be allowed to register a handler for a given command op-code value.

When disconnecting a component, the *deinstallHandlers* method is called, in response to which the component calls *deinstallHandler* on the provided *CommandDispatcher* implementation for each command type that it handles. Components must guarantee that the *deinstallHandlers* will deinstall any and all command handlers that the *installHandlers* method did or could install. Successful run-time modification depends on this property of components.

In operational mode, commands are received by the *CommandDispatcher* and immediately validated, using the *CommandParser* provided by the component that handles the command. If the command parses successfully, it is placed on the handling component's *Queue*.

Our command processing interfaces make a few assumptions that are driven by our FSW design: the direct use of the *Queue* interface as the mechanism of passing commands reflects a design choice that might not be appropriate in other similar architectures. Similarly, our design choice to put command processing on separate threads drove us to include a *Scheduler* implementation in the parameter lists of the operations of the *CommandProcessor* interface: this allows the component to obtain a *Queue* for command reception and schedule a task for processing that *Queue* at the same time.

Scheduling Interfaces

Figure 5 shows the interfaces and classes involved in making and breaking scheduling connections among components. Our architecture features three kinds of thread run logic interfaces: *Periodic*, *QueueReader*, and *Sporadic* (readers familiar with [3] will recognize some of these interface names. Our interfaces are loosely similar to the corresponding interfaces in the RTSJ.)

As with the command processing interfaces, the *Component* base class provides a no-op implementation of the *TaskOwner* interface, and only components that need to schedule threads need override that interface. At connection time, *scheduleTasks* is called on the component. The component in turn, calls *addToSchedule* on the given *Scheduler* interface implementation for each thread that the component wants to schedule.

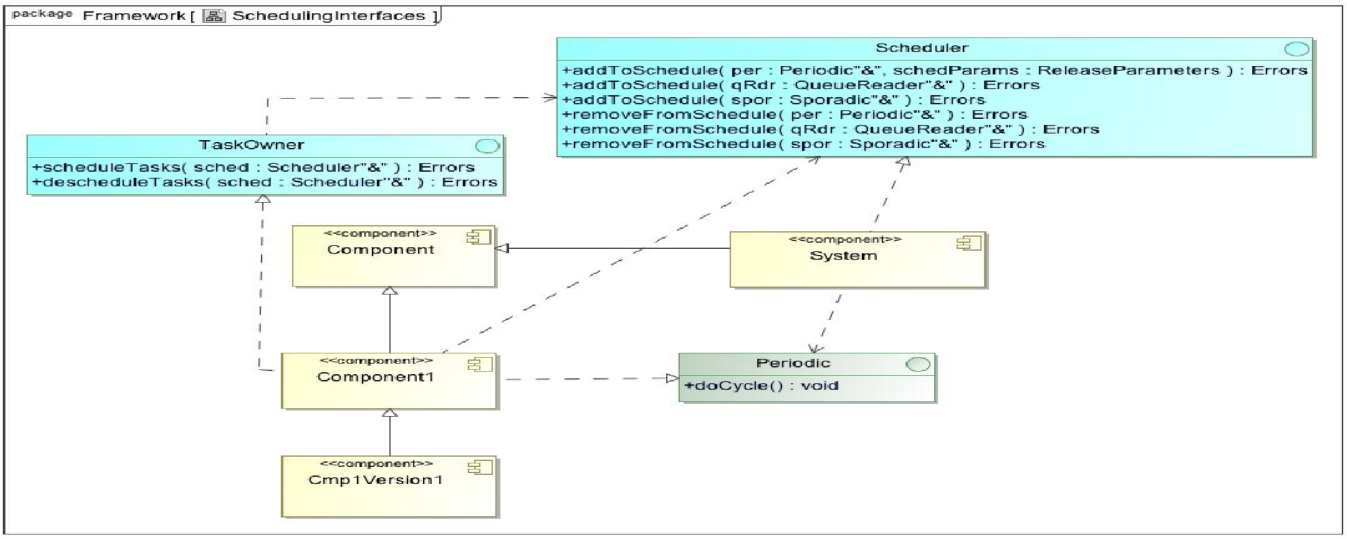

Figure 5 - Scheduling Interfaces

At disconnect time, the *descheduleTasks* operation is called on the component, which in turn calls *removeFromSchedule* for each task that it scheduled previously.

Only *Periodic* tasks actually receive CPU time from the *Scheduler* implementation in operational mode. However, the *Scheduler* implementation in our design provides the services of creation, initialization, and finalization of the thread objects that actually run the given interface implementations. The *addToSchedule* operations create and initialize threads for the given run logic object, and the *removeFromSchedule* operations finalize and destroy that thread object. These interactions are not shown in our diagrams. As with any of these architectural interfaces, the modification process depends on the *removeFromSchedule* operation undoing any of the effects of the *addToSchedule* operation.

The operation *addToSchedule* for *Periodic* tasks allows the component to specify the period, in clock ticks, and the offset from the first tick, at which the *Periodic's doCycle* operation is to be invoked. The implementation of *Periodic* need only override that one operation.

In our design, the Scheduler interface implementation is provided by the System component.

Telemetry Processing Interfaces

Our architecture supports two main styles of telemetry production: active and passive. As shown in Figure 6, a component that wants to produce telemetry must override the *Component* base class' trivial implementation of the *TelemetrySource* interface. In another example of interface implementation hand-off, the parameter list of each of the operations of the *TelemetrySource* interface consist of an implementation of the *TelemetryManager* interface.

Components that are active producers of telemetry must implement the *register/deregisterActiveProducers* methods. Again our design choice to use a *Queue* for passing telemetry from active producers to the Telemetry component shows in the interface: the *QueueAttributes* parameter to the *registerActiveProducer* method is used to create a *Queue* implementation, returned by the call, to which the active producer sends its data, assumed by the *TelemetryManager* to be in the form of objects of a class that is an implementation of the *TelemetryItem* interface.

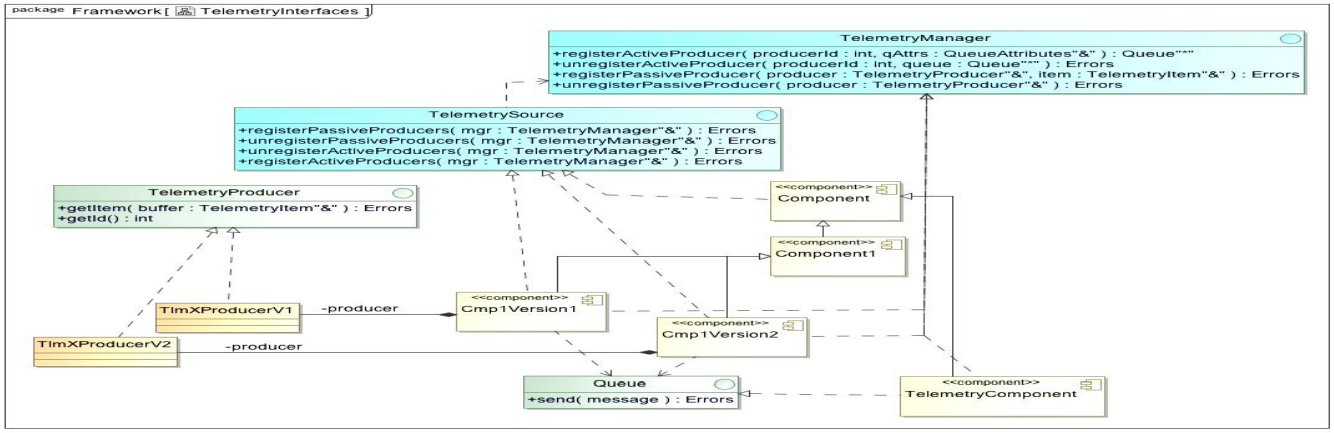

Figure 6 - Telemetry Processing Interfaces

Our *TelemetryManager* implementation supports two styles of queue draining on behalf of active producers: a "keep all" style, and a "keep only the latest" style. In the case of an active producer of error notifications, the "keep all" style is used, while for more typical producers of subsystem engineering telemetry, the latter style is appropriate.

Passive producers provide the *TelemetryManager* with an implementation of the *TelemetryProducer* interface. During operations, it is the *TelemetryManager* that decides when to query the producer object.

At connect time, the two register operations on the component are invoked. The component in turn calls the appropriate register operation on the *TelemetryManager* interface. At disconnect time, the two unregister operations are invoked on the component, resulting in calls to the unregister methods on the *TelemetryManager* implementation by the component.

The Linkable Interface

The architecture framework allows the definitions of other interfaces among components in addition to the command, scheduling, and telemetry interfaces provided by the architecture framework. The framework provides support for other interfaces using the *Linkable* interface. As we see in Figure 7, the *Linkable* interface gives the component that implements it the opportunity to link to, or unlink from, the component given as the argument to the call.

As with the other architectural interfaces, the Component base class provides a trivial, no-op implementation of the Linkable interface. Components that need special connections to some other component must override that implementation. Implementations of the methods of Linkable typically use the RoleFiller interface to decide whether the component provided as the argument can provide an interface that this component needs.

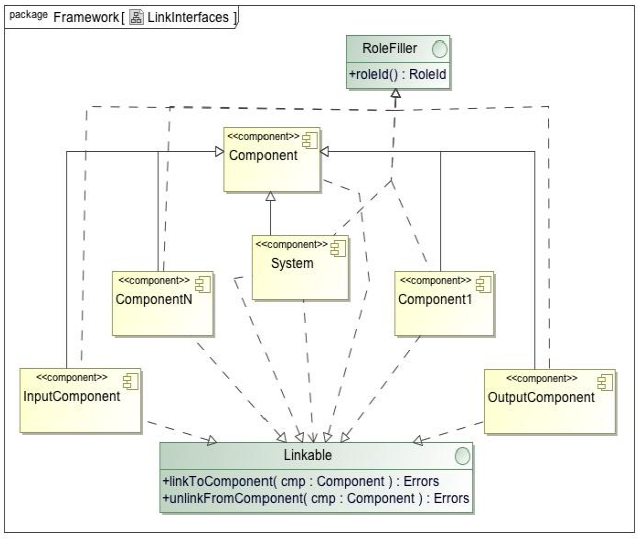

Figure 7 - The Linkable Interface

In the Aquarius design, there are several cases of the *Linkable* interface being used to provide special connections between components. For example, we have a component that provides an interface to the 1553 bus. The Telemetry component needs that interface, so it implements the *linkToComponent* operation with logic to see if the role of the given component is the 1553 interfacing component, and if so to ask it for the interface implementation. In another example, a science data formatting and storage component needs to connect to a component that is responsible for communications with the radiometer in order to get an implementation of an interface that provides that instrument's data.

As with the other architectural interfaces, the *unlinkFromComponent* operation must be the complete inverse of the *linkToComponent* operation.

The Component Class Hierarchy

Having gained some insight into the architectural interfaces, it's helpful to see an overview of the component class hierarchy, shown in Figure 8. All four of the component-provided architectural interfaces – *CommandProcessor, TaskOwner, TelemetrySource*, and *Linkable* – are implemented by the *Component* base class, in a trivial, no-op way. This ensures that these interfaces can be exercised on any component instance. As described in the previous paragraphs, components that need to participate in architectural activities, such as command processing, must override the appropriate set of these four interfaces.

This diagram also shows the component classes *Component1* and *ComponentN*. These classes serve as examples of classes that establish the behavior of specific roles (in this example role 1 and role N, whatever those might be.) Suppose for example that role 1 is the role of a component that provides command and control of some specific subsystem, such as the Aquarius radiometer. Then class *Component1* would implement nearly all of the behavior needed to fill that role. However, *Component1* is an abstract class because it fails to provide an implementation of the interface *Versioned*, which is meant to provide a notion of versions of the implementation of a particular role.

The class *Cmp1Version1*, derived from *Component1*, implements *Versioned* and is thus a concrete class, the class that represents the launched version of the component that fulfills role 1. The class *Cmp1Version2* represents the class that would be uplinked in order to replace that component in flight. The class *Cmp1Version2* could do as little as merely implement the *Versioned* interface differently, or it could override many of the classes and operations defined in *Component1* and its constituent classes.

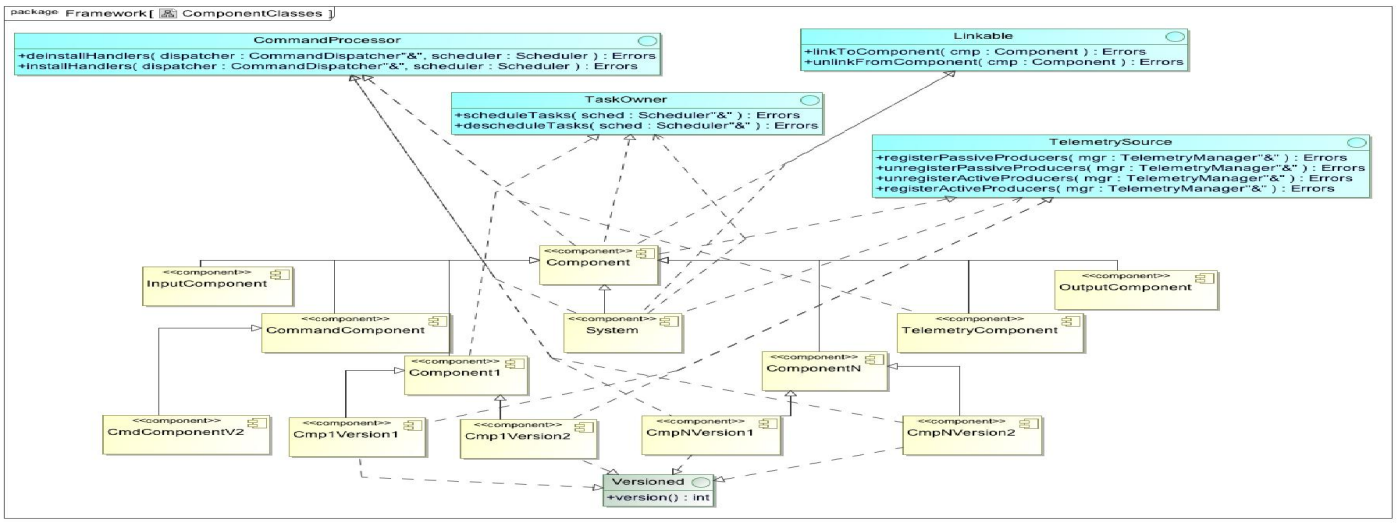

Figure 8 - The Component Class Hierarchy

Architecture Management

The run-time management of the architecture is performed by the System component. Referring to Figure 9, we see that the framework class *Architecture* provides most of the operations needed to establish or change the run-time architecture of components. The only abstract operation in the framework's Architecture class is *createComponents*. That operation is abstract because the framework cannot know what the set of components that are to be part of the architecture is, nor the concrete types of the components. The application-level class Architecture provides this information by implementing the *createComponents* operation.

The System component creates an object of type *Application::Architecture* and uses it to manage the architecture, both during system initialization and finalization, and during run-time modification. In fact, the process of modification consists of snippets of initialization and finalization behavior.

Figure 9 introduces another architectural interface, a sort of "meta-architectural" interface called *ProviderAnnouncer*. This interface lets the architecture object query the set of components in the architecture to discover providers of the architectural interfaces *CommandDispatcher*, *Scheduler*, and *TelemetryManager*. If it discovers providers, it can then use the interfaces on the components that require these providers. For example, if some component in the architecture provides an implementation of *CommandDispatcher*, only then can the Architecture object invoke the methods of the *CommandProcessor* interface on the components.

With that, we are in a position to step through the process of run-time modification.

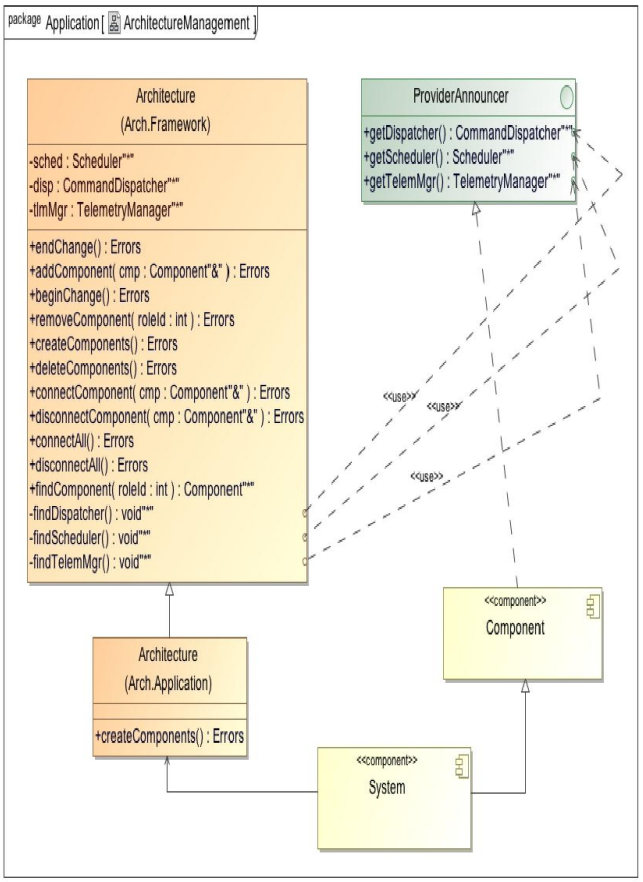

Figure 9 - Architecture Management

3. FSW Modification Process

As depicted in Figure 10, modification of the running flight software begins with the operations team identifying some reason for the modification. The reason might be to fix a FSW bug discovered in flight (though the presence of bugs in our FSW is *highly* unlikely), or to adapt to degrading or

failed hardware, or to take advantage of an unforeseen science opportunity, or other situation.

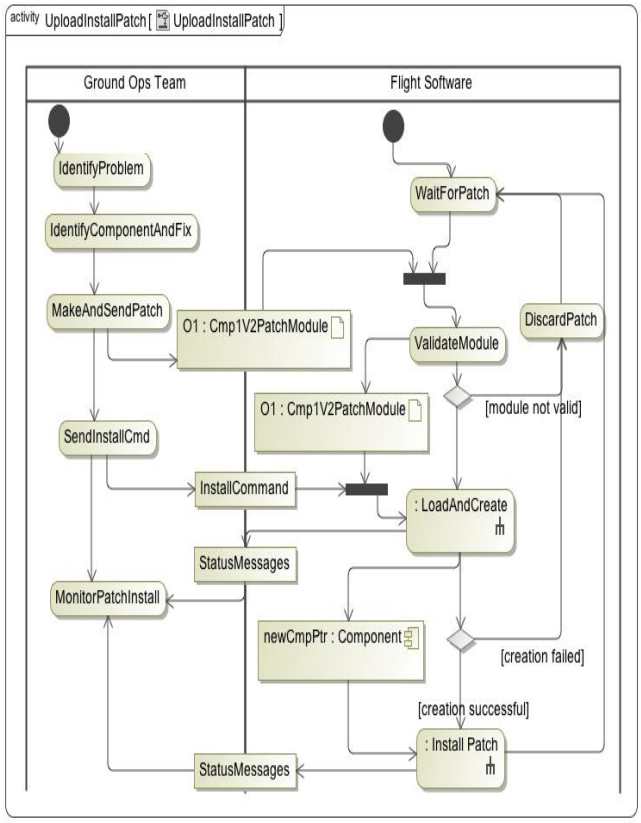

Figure 10 - The Modification Process

Whatever the reason, the team must identify which component in the FSW is to be modified, and the behavior to be changed, or decide to install a new component, and identify the new behaviors desired. Then the patch must be implemented, by defining and implementing a new sub-class of the component to be modified, and possibly also new subclasses of the constituent parts of the component.

The patch and patch file are depicted in Figure 11. In addition to the new classes needed to modify the component's behavior, the patch contains a class that is derived from the class *ComponentFactory*. In the diagram, the example given is class *Cmp1Version2Factory*. The patch file will also contain an object of type *Cmp1Version2Factory*, and an integer variable *sideEffect,* described below.

The patch is compiled and linked. The patch file itself, in Executable and Linking Format (ELF), contains the machine code and data segments for the new code only. It can refer to objects and functions that are already part of the code repository in flight, but it must not redefine those objects or functions (that would result in multiply-defined symbol errors at load time).

Also, when compiling and linking the patch file we explicitly prevent the instantiation of C++ templates, except for those that are defined only in the modified component class and not in any of the classes that are already part of the in-flight code base. This keeps the patch file smaller and prevents wasteful uplinking and in-flight storage of redundant template instantiations.

The patch module is then converted into a command script and uplinked to the flight system. The uplink process may take significant time and be done over multiple uplink passes. The first command in the script informs the FSW whether this module represents a replacement patch, or a new patch, and, in the case of a replacement patch, the role ID of the component to be replaced.

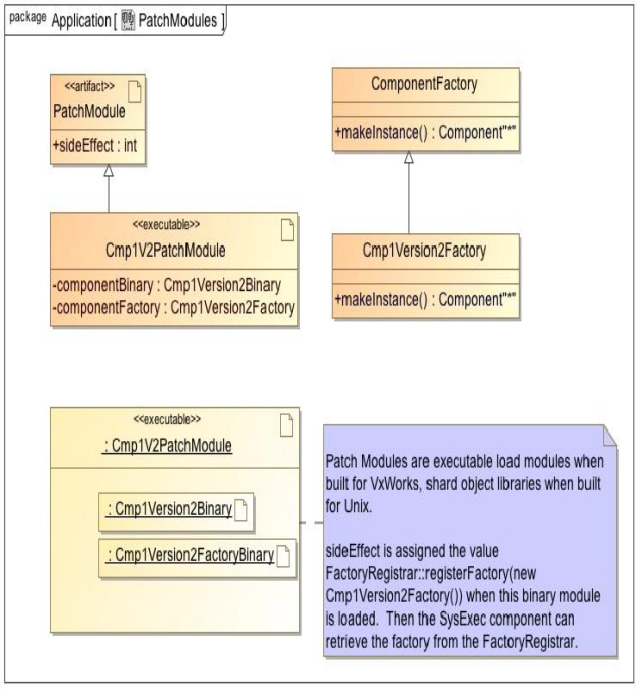

Figure 11 - Patch Modules

The FSW assembles the pieces of the module in a RAM-hosted file system as a file. The command script ends with a special command that tells the FSW that all pieces of the module have been received and provides the checksum of the module. Then, in the *ValidateModule* step in Figure 10, the FSW computes a checksum over the entire module, compares it to the ground-provided value, and rejects and discards the patch if the checksums are not equal.

If the module is valid, and when the ground sends an *installPatch* command, the FSW performs the *LoadAndCreate* step shown in the diagram. If that succeeds, the FSW has created (in the sense of C++ - a dynamically-allocated object) an instance of the new component class (class *Cmp1Version2* in the example shown in Figure 10), and it can then proceed to install the new component instance into the architecture, shown as the *InstallPatch* step in Figure 10.

Zooming in on the *LoadAndCreate* step, we see in Figure 12 that the System component calls the OS-provided load function, the argument being the name of the ELF patch

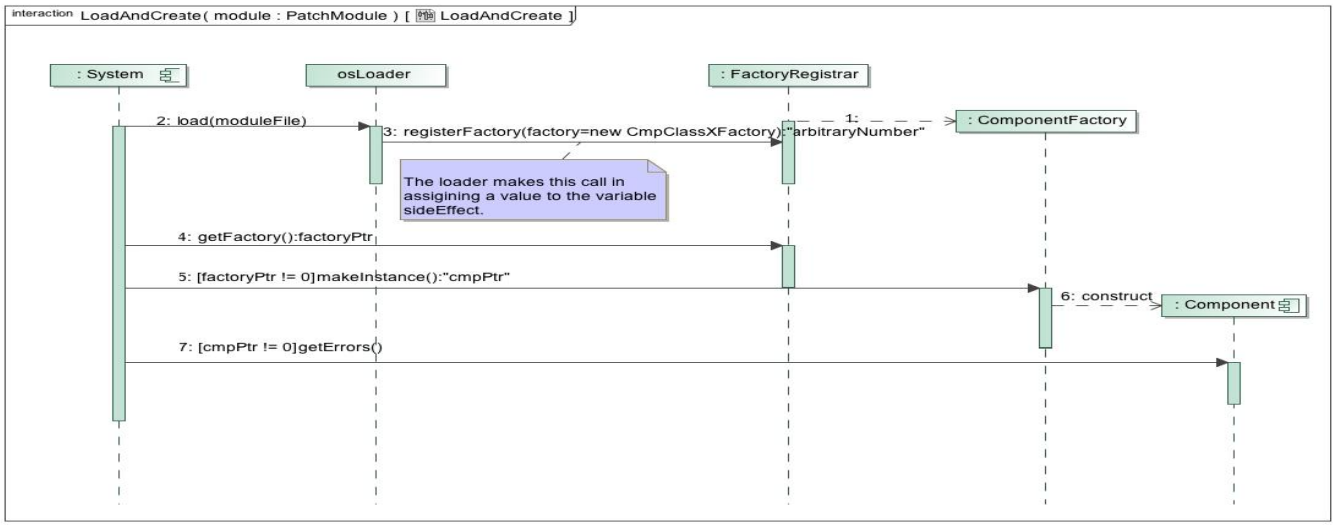

Figure 12 - Loading and Instantiation

module file. The OS loader then reads the file, resolving references made by the code in the file to functions that are already part of the code base, and also initializing static variables defined in the load module. In assigning a value to the variable *sideEffect*, the loader calls the static function *registerFactory* on class *FactoryRegistrar*, which has the effect of registering the module's component factory (of type *Cmp1V2Factory*), with the *FactoryRegistrar*. Thus the factory object becomes available to the FSW via the *getFactory* function.

Having obtained the factory instance, the FSW calls *makeInstance* on that object to create an instance of the new component. The makeInstance method of *Cmp1V2Factory* runs the constructor of the class *Cmp1Version2*. The constructor may encounter errors, which the FSW must check for by calling the *getErrors* method on the new component. Every step of preparing for and installing a patch is checked for errors, though we do not show all of that detail in our diagrams.

After these steps, the FSW is ready to install the patch, which means connecting the new component into the architecture and then letting it run. This process is shown in the sequence diagram in Figure 13.

Figure 13 - Installing a Patch

The first message of this sequence is the *pause* operation on the System component. That operation disables the two interrupts that drive the FSW, the 100 Hz external interrupt, and the 1553 bus interrupt. This allows the FSW, in the context of the System component's architecture management thread, to safely change data structures that would normally be accessed from different threads in operational mode. These data structures include the *CommandDispatcher's* map of op-code to parser and handler, the *Scheduler's* lists of run logic objects and associated Thread objects, and the *TelemetryManager's* lists of *Queues* and passive producer objects.

The second message in the sequence, the *beginChange* operation on *Architecture*, has the effect (not shown) of announcing to every component in the architecture that architectural change mode has been entered. This tells the components that they must be ready for calls on their architectural interface methods, and also that the dynamic memory allocation (C++ *new*) is now allowed (dynamic allocation is forbidden in operational mode, a constraint that is automatically enforced in the Unix build of the FSW by intercepting *malloc* calls.)

If we are replacing a component that fills one of the known roles, we must first find the old component and remove it from the *Architecture's* map of component instances, and then disconnect it, which is accomplished by the *disconnectComponent* call on *Architecture*. That method calls all of the unregister, deinstall, and unlink calls on the architectural interfaces of the component being disconnected, as well as the *unlinkFromComponent* method of all components in the architecture with the old component as argument. This assures that, after these steps, no object in the architecture is referencing the old component or any of its constituent objects. Error checking is made with each step, and recovery operations undertaken if errors occur, though these operations are not shown in our diagrams.

Having disconnected the old component instance (if this is a replacement patch), the sequence continues by adding the new component instance to the *Architecture's* map of component instances, and then calling the *connectComponent* method on the *Architecture* object. Figure 14 shows the sequence of operations made by the *connectComponent* method. Here we see the architectural interface implementations of the new component being exercised, as well as the *Linkable* interface on all of the components of the architecture. When the connection sequence is finished, the *endChange* method is called, which announces to all the components that the system is transitioning from architecture change mode operational mode.

The final step is to invoke the *resume* method on the System component, which re-enables interrupts and starts the system running in operational mode.

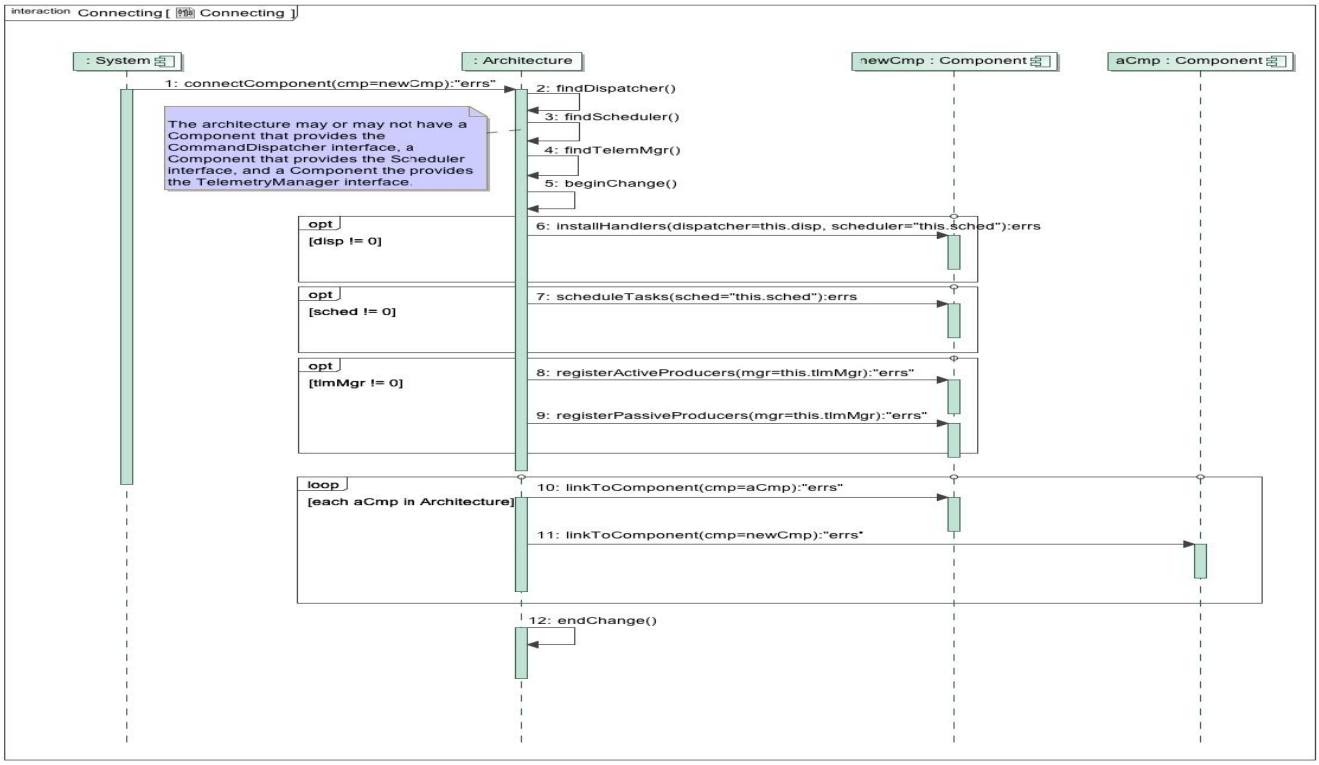

Figure 14 - Connecting a Component

During the entire uplink, validation, load and install sequences, the FSW emits telemetry to keep the ops team informed of the progress and status of the modification operation.

Recovery from a Failed Patch Process

Any step of the patching upload and installation process can fail, and the FSW must be able to handle these errors. We have not shown error handling in the sequence diagrams in order to keep them simple. Following is a discussion of the errors that can happen and the autonomous recovery steps that the FSW takes in response to each.

Errors during uplink of the patch are detected by sequence numbers in each message and by a checksum over the entire patch. A corrupted patch will be discarded, and the patch operation aborted.

Errors in the module itself, such as undefined symbols, or a failure of the module to register a component factory at load time, are detected by the FSW, and it discards the module and aborts the patch operation.

If the constructor of the replacement or new component detects an error, the new instance is deleted, and the patch operation aborted.

The next step for a replacement is to remove the component to be replaced. This is done using the unlinking and uninstalling operations of the architectural interfaces. If any of these calls fail, the FSW attempts to re-connect the old component to the architecture, and aborts the patch operation.

If the old component is successfully removed, the next step is to link the new component into the architecture, using the linking and installing operations of the interfaces. If any of these operations fail, the FSW takes the disconnection steps with the new component, and then re-connects the original component (which is not deleted unless no errors occur) back into the architecture.

The FSW reports success or failure of each step to give the ground insight into the process. If any of the errors occur, the architecture could be in a partially-functional state: e.g., unable to handle certain commands, and the ground would need to carefully evaluate whether a reboot might be prudent at that point.

Whenever the FSW attempts backup steps to recover from an error, it finishes the patch operation by putting the architecture back into operational mode and re-enabling interrupts, regardless of whether or not the backup steps themselves encountered errors. This is safe in Aquarius' case because the FSW cannot damage the hardware, and so we can allow the FSW to stay running in a degraded state in the hope of getting information out to the ground about what went wrong.

In the next section, we describe how we test all of the failure scenarios we've just described.

4. VERIFICATION

The FSW modification process that we've described is highly automated, and so involves many steps and conditional branches in FSW logic. And since it involves disabling the interrupt that allows the FSW to communicate through the 1553 bus, it could leave the system in a non-responsive state if something went wrong. [3] Therefore it was essential that we verify the correctness and robustness of this process as exhaustively as possible.

We approached this task with several techniques, chief among them testing, but also with code checkers, code coverage analysis, and detailed code reviews. Our testing included component- and architecture-level white box testing, automatic generation of long sequences of modification operations with injected faults, and system-level tests that replace every patchable component and introduce several entirely new components. We insisted on code coverage of 100% for the parts of the System component and other classes directly involved in architecture modifications.

Component-Level White Box Testing

On the level of an individual component, we needed to verify that each component class implemented its architectural interfaces correctly, and also that the methods implementing the architectural interfaces responded appropriately to faults that could occur. For each component class, we wrote a stand-alone test program that exercised all of that component's architectural interfaces, and caused them to encounter every error that they could encounter.

To accomplish this, we developed special test component classes called *fault injectors*, depicted in Figure 15. Consider for example the operation *registerActiveProducerFaultIn* on the fault injector class *FaultInjectorTlmManager:* the *callCount* parameter specifies the number of calls to the method *registerActiveProducer* that will occur before that operation will return a non-zero error count, at which time the component attempting to register the producer is forced to handle that error. The call count logic is necessary because most components make more than one call to a register or install operation, and we needed to make each one of those distinct calls produce an error.

A component-level white box test program creates instances of each of the fault injector classes, and then runs a series of

[3] We considered adding a watchdog thread to re-enable the interrupts after a time if the thread on which the architecture modifications were being done never re-enabled interrupts, but decided that it was not necessary for the Aquarius mission, because there is a watchdog on the spacecraft, and the consequences of a reboot are not dire.

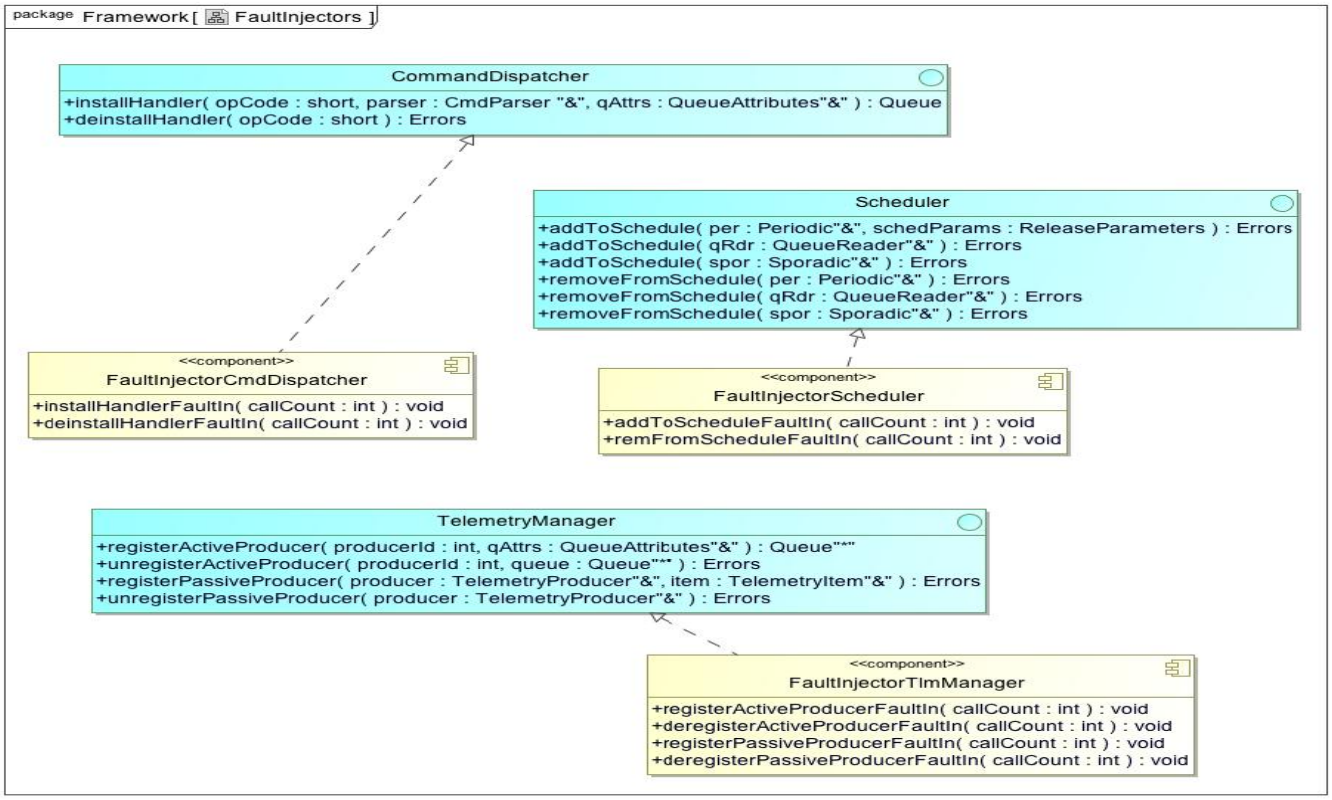

Figure 15 - Fault Injector Classes

tests in which the fault injectors' methods are used to force the component to handle every possible error that it could encounter.

Automatic Test Scenario Generation

We developed a patching test harness with the goal to determine the robustness of the patching system with respect to faults while validating the connection and disconnection logic for each component. Our approach gives the ability to specify combinations of patchable components and fault injections and perform software patching under these fault conditions.

The test harness enables the user to choose permutations of components picked from the set of patchable components, where repetitions are allowed. Patching the same set of components with different orderings puts in evidence any dependency relations between connection and disconnection operations. These dependencies can be in the form of memory allocation and de-allocation issues, or unsafe assumptions on the life of a shared object.

The fault combinations are generated as follows: For every method specified in the fault injection component (see Figure 16), we specify a *fault* or *no fault* condition. The test harness generates a user-specified number of test cases, where each test case represents a random set of faults. The set of patchable components are then patched with those faults. Thus thousands of patch operations are made in a randomly-selected order.

We specialize the *FaultInjector* component (see Figure 16) with a patchable component type. A call to a *FaultInjector* component method results in a call to the specializing component's corresponding method. This method is called with or without the inclusion of a fault, depending on whether a fault was specified for the particular method a priori.

The inclusion of faults during patching tests the ability of the patching system to safely recover from faults associated to the initialization and finalization of flight software subsystems. It puts in evidence the dependencies between the methods of a particular component.

For example, if *installCmdHandlers* had a fault and it was called before *linkToComponent*, can *linkToComponent* execute correctly given a fault in *installCmdHandlers*? These dependencies are extracted more readily by executing different fault injection scenarios. As a result, the flight software system's resilience to unsafe patching operation is verified.

System-Level Testing

We have a suite of system-level patching tests in which we replace every patchable component and install several new components. The suite includes successful patch operations, as well as all of the failures that can be generated when running the system as a black box, with only the flight interfaces (the 1553 bus and the high-speed serial bus).

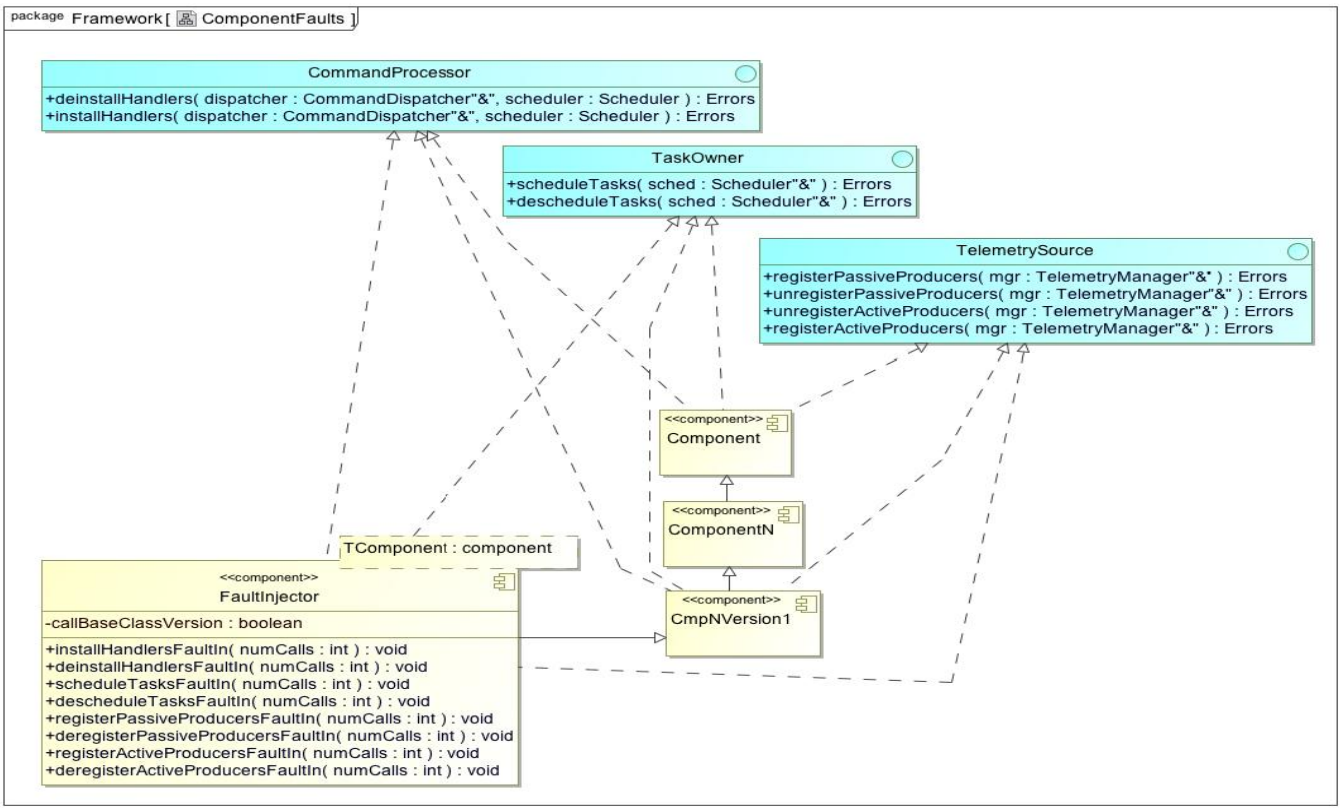

Figure 16 - The Fault Injector Component Template

To date, we have run these tests in two different environments: one in which the only hardware we have is a breadboard PowerPC that has a serial port, and a separate board that provides Ethernet connectivity. In this configuration, software simulations for the other boards that make up the instrument's command and data subsystem are run as part of the FSW image (by constructing the architecture with simulation versions of the components that interface to these boards.) The 1553 and serial buses are simulated by Ethernet socket messages.

A variant of this stand-alone environment is running the FSW on a Sun workstation. All OS-dependent constructs are wrapped in C++ classes, and the differences among OS's are hidden in the implementation of these classes. This technique enables us to run a FSW image on the Sun that exercises at least 90% of the code in the FSW. In particular, none of the patching logic is OS-dependent (except for the OS's loader, and some subtleties of thread behavior). Patch modules in the Sun configuration are shared libraries.

The second environment is in the integrated command and data subsystem. In that environment, a modified version of the same commanding tool we use for simulation of the 1553 bus in the stand-alone environment sends commands to a real 1553 bus. So we can use the same command scripts in either the stand-alone environment or the integrated command and data subsystem environment. We've run these tests on engineering models as well as the flight version of the subsystem.

As of this writing, the complete flight instrument has not yet been integrated, though it will be within a few months. When it is, we will run patching tests on that system.

5. APPLICATIONS

To date, the run-time modification capability has been extremely useful in testing situations. It has enabled tests on a system level that would normally be possible only in white-box context, allowed rapidly prototyping design modifications and problem fixes, and served other utility purposes. Following is a brief discussion of some of the applications of this capability to date.

(1) Writing the EEPROMs: Late in the development, we realized that writing the EEPROMs using a patch instead of the traditional serial port-based EEPROM writing application would save us from having to modify the flight hardware configuration to enable the serial port if we wanted to write the EEPROMs after the flight C&DH had been assembled. (This would also open the possibility of writing the EEPROMs in flight, though this is not planned.) Thus the EEPROM Writer component was developed to write a flight software image to either (or both) of the two on-board EEPROMs. This component is not part of the original flight software component configuration. It is itself uploaded as a new flight software patch. There are two on-board EEPROMs; one on the CPU board and

the other on COM board. The EEPROM Writer component is capable of writing to and verifying of either EEPROM, selected via ground command. The EEPROM Writer component, of class *EepromCmp*, consists of an *EepWriter*, and *CmdHandler* classes, as shown in Figure 17. During the connection process, the *installCommandHandlers* method registers all the new ground commands needed for uplinking a complete FSW image, with unique op-codes and their associated command parsers. The *scheduleTasks* method adds the *EepWriter* object to the schedule as a new periodic task with pre-selected priority. The EEPROM writes must be timed carefully, and this *Periodic* runs at 100 Hz to accomplish this. Once installed, the EepWriter's *CmdHandler* object wakes up on a series of ground commands to upload a new flight image. The image is written to one of the EEPROMs in chunks during periodic calls from the *Scheduler* to the *EepWriter*.

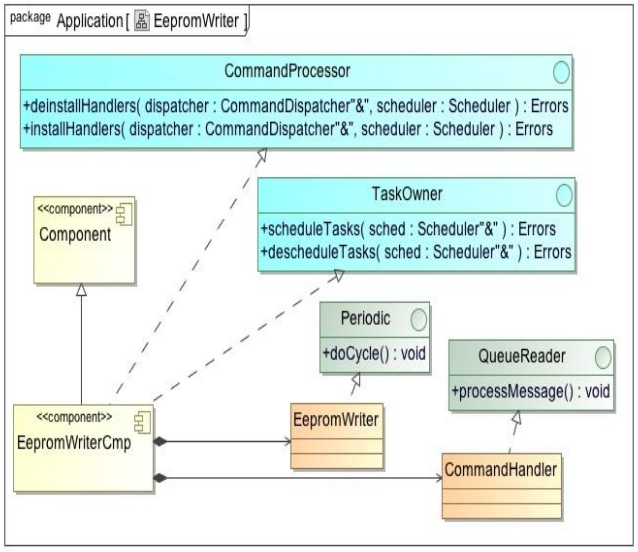

Figure 17 - The EEPROM Writer Component

(2) CPU stress test: we uplink a new component whose sole purpose is to waste CPU time at a high priority, and to progressively use more of it as time goes on. We want to make sure that the performance of the FSW degrades in a predictable way. This patch is called the *CpuHog*.

(3) Prototyping a different design of 1553 driver. We wanted to optimize the driver, but we wanted to try it out before changing the baseline and overwriting the EEPROM, so we made a patch of the component that contains the driver. After debugging the driver, we decided to incorporate the new design into the baseline.

(4) We had to make a change in the FSW timing of reads from an analog accumulation register because the board required more time than its specification said it did. We made the modifications necessary in a patch of the component that contains the interface to that register, and, after some experimentation and modifications (over several patch uploads), we settled on the best fix and made it part of the baseline.

(5) In thermal testing, we found that a serial bus driver chip became very noisy above a certain temperature, and had to change the FSW to handle this, which included disabling an interrupt and other changes. Under extreme time pressure, we quickly put the fix into a patch, tested it, and then incorporated it into the baseline. Doing the patch let us test the fix in the thermal chamber *before* having to write the EEPROMs with a new edition of the FSW.

(6) Fault injection in system-level patching tests: we have several components that intentionally fail at different steps in the patching sequence so that we can test the FSW's response to these failures.

(7) Cleaning up after and replacing a suspended task: on one of our system test scenarios, we intentionally peek an invalid address, which causes the thread that handles the peek command to get suspended. We don't want to reboot, so we uplink a replacement patch for the component containing that thread. The finalization process of the replaced component deletes the original thread, and installing the replacement creates a new one. In this case, we install a new instance of the very same version of the component. This could avoid a reboot in flight.

6. Conclusions

It is natural to question the necessity of the kind of flexibility that our modification technique provides. After all, once launched, a system can hardly change so much that it would need major behavior modifications. With crossed fingers, we hope this will hold for Aquarius, and that we will never have to patch the FSW in flight. But should the need to change our FSW in flight arise, in however unpredictable a way, we are confident that our modification capability gives us ample ability to adapt. Moreover, there are a number of advantages to our patching method over the usual poking of machine instructions directly into memory:

(1) The flexibility of our patching process greatly facilitates test and development, as our list of applications in the previous section demonstrates. We have institutional requirements that FSW be readily testable, and that it allow the efficient diagnosis of unexpected conditions and faults. We think our patching approach has allowed us to meet these requirements in a more complete way than has been done before.

(2) The relative ease of patching our approach enables by allowing the development of patches at a higher logical level than patching one function, and the

automation of the entire process, has allowed us to use patches more widely and frequently than we could easily do with traditional patching, and this has helped verify this capability much more extensively than we might have otherwise been able to do.

(3) Delegating the loading of code and placement in RAM of the code to the OS frees us to test the same patch on several machines, with different memory layouts, FSW versions (though patches must be compiled against exactly the same versions of FSW libraries as the running version is), or even with different operating systems. This has allowed us to run many, many thousands of patching operations as part of our testing program, and again has contributed to the reliability and robustness of our patching process.

(4) Our modification process allows us to significantly modify the behavior of the FSW, well beyond the behavior of an individual function. This will enable us to handle fault conditions or other unexpected conditions that require extensive and complex FSW changes, without having to reboot.

Future Work

There are a number of areas in which we'd like to apply this capability, and to expand it and its application.

(1) Larger Systems: This approach to FSW modification has been used to date only for an instrument, Aquarius, albeit a fairly complex compound instrument. We feel that this approach would serve a spacecraft flight system very well, and we hope to get an opportunity to try it. The robustness of the technique could have to be ratcheted up in this context, but that is entirely doable.

(2) Software Fault Protection (FP): Often software errors, or other fault conditions that result in unexpected conditions in the FSW, result in a suspended thread. In Aquarius, the FSW detects and reports suspended threads, but takes no automatic corrective action. This does allow the ground to recover without a reboot. However, it would be a natural extension of the logic to make the FSW automatically finalize and re-initialize the component containing the suspended thread. This could allow the FSW to recover from a suspended thread condition in a very small number of seconds, rather than the minute-plus times typically associated with the reset or reboot of a flight system. This in turn could significantly increase the robustness of a flight system in short, time-critical phases, such as orbit insertion, in which there is no possibility of ground intervention because of distance. Also, avoiding a reboot of a flight system could save many hours of ops. team work in bringing the system back into mission mode.

(3) Fault Diagnosis: Being able to automatically and rapidly launch new or significantly different behavior in the FSW could be used to rapidly increase the level of visibility into the state of some subsystem or piece of hardware. For example, if the FSW detected an unexpected condition in some piece of hardware, it might automatically launch a component that performed special monitoring of that hardware, or that accepted special commands affecting that hardware. Even in our current implementation, it would allow the ground to uplink and install special diagnostic components in such a situation.

Overall, we believe this type of approach will become widely used in flight software in the future.

ACKNOWLEDGMENTS

We would like to thank our colleagues on the Aquarius project, in particular the Instrument Engineer, Dalia McWatters, and the command and data subsystem engineers, Mimi Paller (lead), Andy Berkun, Mark Fischman, and Craig Cheetham for their excellent work in quickly and unerringly producing clear interfaces, requirements, and subsystems: this made the FSW team's job much easier. We also extend our gratitude to our software-engineer colleagues who were involved in the Mission Data System software development at JPL: David Wagner, Kirk Reinholtz, Nicolas Rouquette (an important proponent of component architectures at JPL), and Tim Canham; many of their ideas have contributed to the Aquarius FSW design. Finally, we thank our colleagues Wafa Aldiwan and Cin-Young Lee for valuable reviews of this paper.

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[1] A. Murray, M. Shahabuddin, "OO Techniques Applied to a Real-time, Embedded, Spaceborne Application", ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2006 proceedings.

[2] NASA's Aquarius mission website: http://aquarius.gsfc.nasa.gov/

[3] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, D. Holmes, M. Turnbull, R. Delliardi, A. Wellings, "The Real-time Specification for Java" (RTSJ), Version 1.0.2.

[4] M. Fowler, "UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)", Addison-Wesley, Sept 15th, 2003

BIOGRAPHY

Alex Murray is a senior software engineer with the Jet Propulsion Laboratory, California Institute of Technology. He has most recently led the development of the flight software for the Aquarius instrument. He has led and done software development for flight, ground, and simulation software for missions and for technology development projects at JPL. Previously he led and developed software for a variety of projects at TRW (now Northrop-Grumman), and he served as a system engineer for the European weather satellite agency, Eumetsat, as well as software engineer for the Dresdner Bank in Frankfurt, Germany. His experience includes embedded and flight software development, prototype and research development, OS development, AI, analysis and simulation tools, science and image processing applications, business and GUI applications, and databases. He holds BS and MS degrees in mathematics from The Ohio State University.

Mohammad Shahabuddin is a Senoir member of the Technical Staff at the Caltech Jet Propulsion Laboratory. His contributions to NASA include design and development of Aquarius instrument flight software, Mission Data System (MDS) flight framework, multi-mission simulation framework and software simulators for Galileo, Voyager, and Cassini compute data hardware. Object oriented principles were applied in the design and development of Aquarius software, using UML design tools and C++ language. He and his team earned NASA award for developing bit-level High Speed Simulator for Cassini Spacecraft used as a testbed for command and sequence validation. He received his M.S. degree in Electrical Engineering from California State University at Long Beach. He did his undergrad in Electrical Engineering from Engineering University Peshawar, Pakistan.

Vanessa D. Carson is a member of the technical staff at the Jet Propulsion Laboratory (JPL), California Institute of Technology whose experience ranges from research in model-based software engineering to applied design and implementation of spacecraft and rover software systems. Ms. Carson is currently a Ph.D. student in the Control and Dynamical Systems department at the California Institute of Technology (Caltech). She also holds a B.S. Cum Laude and M.A. in Mathematics from the University of Southern California. Her recent projects include the design and implementation of a planning system for the DARPA Urban Grand Challenge with Team Caltech and the implementation of an onboard patching software system for the Aquarius Mission at JPL.